
UNIVERSITÀ DI PADOVA FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

# PariCore

# Refactoring del Timer

Relatore: **Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi**

Correlatore: **Egr. Ing. Michele Bonazza**

Laureando: **Luca Lazzarini**

# Indice

# Sommario

PariPari è un progetto tramite il quale si vuole realizzare una rete peer-to-peer priva di server, multi funzionale, in grado, grazie al contributo di tutti gli utenti, di fornire molti servizi già oggi disponibili su Internet e di essere aggiornata con nuove funzionalità, qualora vi fosse l'esigenza.

Per raggiungere tale obbiettivo è stato costituito un gruppo di ricerca formato e coordinato da laureandi.

Questo elaborato tratta del lavoro da me svolto all'interno del team di sviluppo del modulo Core, ovvero il cuore di PariPari. Il Core si occupa di lanciare PariPari e i suoi plugin, gestendo nel contempo le comunicazioni tra vari plugin e gli scambi di crediti fra di essi secondo quanto imposto dal modulo dedicato.

Mi è stato affidato il compito di riscrivere la gestione dei PariPariTimerTask, gestiti dal PariPariTimer, per rendere la loro gestione più semplice e cercando al contempo di semplificare la loro gestione futura.

In questa tesi si procederà quindi a descrivere il funzionamento di PariPari e in particolar modo del Timer all'interno del Core.

# Introduzione

**Figura 1: Il logo di PariPari**

Il Timer di PariPari (come quello di Java) non è, come il nome sembra far intuire, un orologio, bensì un controllore che si preoccupa di far eseguire del codice, da qui in poi Task, che i plugin hanno richiesto. Questo perché un plugin può richiedere che un Task venga eseguito ad intervalli regolari a partire da un certo istante, oppure eseguire una tale operazione in modo periodico per un periodo di tempo specificato, come nel caso del plugin `LocalStorage` che ogni 15 minuti salva i propri dati.

La soluzione utilizzata precedentemente per il Timer si basava su classi create ad hoc da chi per la prima volta ha realizzato il Timer di PariPari.

Con l'introduzione di Java 5 e del pacchetto `Concurrent`[1] in esso contenuto, sono stati messi a disposizione del programmatore una serie di classi e interfacce che permettono di semplificare la gestione concorrente di oggetti.

[1] http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html

In particolar modo, la classe `DelayQueue`[2] ha permesso, in un solo passaggio, di creare una coda di priorità che è alla base del funzionamento del nuovo Timer. Essa infatti restituisce automaticamente il prossimo Task da essere eseguito nel momento in cui deve essere eseguito, dopo essere stato inserito nella coda con le necessarie istruzioni.

Ad aggiunta di questo è stata modificata la gestione alla base del Timer, passando da un unico Timer visibile a tutti i plugin ad una costruzione più gerarchica. Ora ogni plugin può creare più istanze di Timer alle quali può assegnare più Task, avendo così un migliore controllo sulle operazioni che verranno eseguite, potendo cancellare o far riprendere i Task in maniera più semplice.

Il tutto però viene ancora gestito da un unico Timer, chiamato `CoreTimer`, non visibile ai plugin, ma solamente al Core, permettendoci così di avere un controllo centralizzato sui Timer potendo conoscere quali plugin abbiano richiesto Timer e in quale quantità.

[2] http://download.oracle.com/javase/6/docs/api/java/util/concurrent/DelayQueue.html

# Capitolo 1

# PariPari

In questo capitolo verranno mostrate le caratteristiche principali di *PariPari*, spiegando la sua struttura e il suo funzionamento in maniera sintetica.

## 1.1 I servizi di PariPari

PariPari è una rete *peer to peer* attualmente in fase di sviluppo presso il dipartimento di ingegneria dell'informazione dell'università degli studi di Padova. Fanno parte del progetto svariati studenti divisi in gruppi di lavoro, ognuno guidato da un *leader* volto a coordinare le attività.
Le caratteristiche di PariPari sono le seguenti:

- Si basa su una rete *serverless*, quindi non centralizzata, dove tutti i membri sono trattati equamente, consentendo di funzionare così a prescindere dal numero di utenti connessi e dalla loro identità senza richiedere di essere continuamente collegati ad essa;
- Presenta un'architettura a plugin, quindi pronta a future espansioni e configurabile a seconda delle esigenze dell'utente;

- Si basa al contempo su un sistema di gestione dei crediti, onde evitare spiacevoli situazioni in cui un plugin utilizza tutte le risorse disponibili senza la possibilità di intervenire.

PariPari si propone principalmente di fornire servizi *peer to peer* attraverso la condivisione di risorse tramite i membri della rete, come ad esempio *File sharing*, *Web Server, DNS, VoIP* e altri, basandosi sempre però su una rete serverless.

## 1.2 La scelta del linguaggio Java

Come linguaggio di programmazione è stato scelto il linguaggio Java per una serie di motivi. Il primo è dovuto al fatto che il codice Java può essere eseguito su qualsiasi macchina e qualsiasi sistema operativo senza la necessità di ricompilare tutto il codice, permettendo così di raggiungere un numero più ampio di utenti senza portare lavoro extra.

Il linguaggio è semplice di sua natura ed è largamente usato dagli studenti, permettendo a tutti di poter essere produttivi in un ridotto periodo di tempo.

Una caratteristica di grande importanza è inoltre la possibilità di lanciare il codice di PariPari direttamente dal browser, grazie a Java Web Start.

Vi è però un grosso svantaggio nell'utilizzo di questo linguaggio, le prestazioni. Essendo un linguaggio ad alto livello, Java può soffrire in certi tipi di operazioni di una minor possibilità di ottimizzazione del codice a basso livello, portando ad un tempo di esecuzione maggiore rispetto a linguaggi più "vicini alla macchina" come ad esempio il C; in ambiti in cui il calcolo si fa intensivo come ad esempio la crittografia, molto usata nelle comunicazioni fra peer in PariPari, questa minor flessibilità si può far sentire maggiormente.

# 1.3 Struttura del client

PariPari non è un unico blocco monolitico di codice, è invece diviso in varie sezioni, d'ora in avanti chiamate plugin, che interagiscono tra loro per offrire determinati servizi, senza conoscere la struttura interna l'uno dell'altro.

L'idea dei plugin si basa sull'utilizzo a *black-box* o a scatola chiusa: non è infatti necessario conoscere come è costituito un plugin per poterlo utilizzare, basta solo che esso rispetti delle specifiche affinché sia possibile comunicare con esso.

La comunicazione non avviene direttamente tra plugin, ma attraverso il *Core,* il cuore dell'applicazione. Suo è infatti il compito di caricare i plugin, soddisfare quando possibile le loro richieste inoltrandole verso il corretto destinatario e controllare che tutti i plugin siano trattati equamente in base ai crediti spesi e ricevuti.

Tralasciando per ora il funzionamento e i compiti del *Core*, del quale verrà discusso nel capitolo successivo, è bene introdurre alcune informazioni relative ai plugin funzionanti in PariPari.

Sta all'utente scegliere quale plugin inserire nella propria copia locale di PariPari, a differenza invece dei plugin presenti nella cerchia interna, rendendo così il progetto espandibile e flessibile. Va aggiunto però che alcuni plugin possono presentare delle *dipendenze*, che qualora non venissero soddisfatte bloccherebbero l'esecuzione del plugin stesso fino al caricamento dei plugin da cui dipende. Il WebServer, ad esempio, necessita dei plugin *Connettività* e *Kademlia* per poter accettare connessioni da altri client e per poter ottenere le informazioni richieste da essi. WebServer verrebbe quindi avviato solo *dopo* che i plugin *Connettività* e *Kademlia* sono stati avviati.

Vi è da fare una piccola distinzione per quanto riguarda i plugin, essi sono infatti divisi in due gruppi, quelli appartenenti alla cerchia interna e quelli appartenenti alla cerchia esterna. I primi hanno permessi speciali per utilizzare alcune risorse specifiche ,quali l'accesso al disco o alla rete, permettendo a tutti quelli che richiedono l'accesso la possibilità di accedere, sempre indirettamente, a tale risorsa. I secondi invece forniscono un determinato servizio appoggiandosi sui plugin della cerchia interna ed eventualmente altri plugin della cerchia esterna.

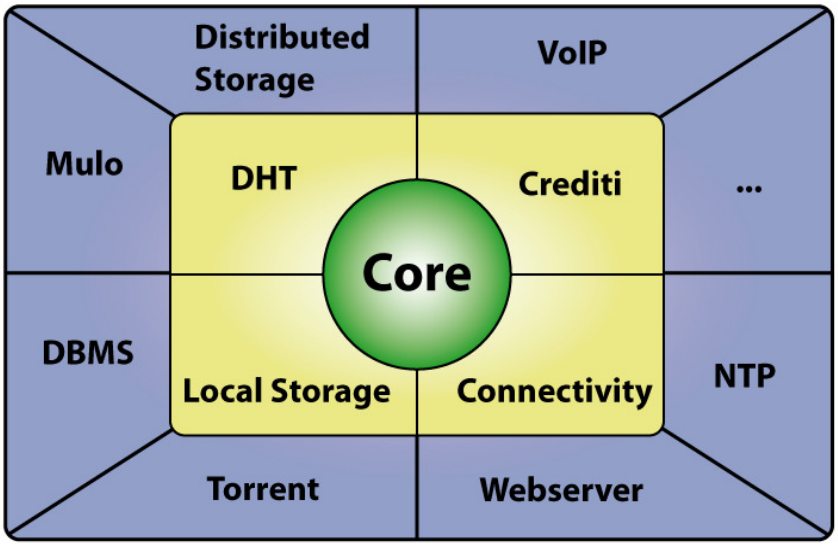


**Figura 2: Schema architettura di PariPari**

# 1.4 I crediti

Data la struttura aperta di PariPari, in cui ogni persona può aggiungere il proprio plugin creato a partire dalle API fornite dal progetto, vi è la necessità di controllare che i nuovi plugin non ostacolino o interrompano l'esecuzione degli altri, rubando memoria o cicli ci CPU inutilmente, sia per una programmazione poco accorta, sia nel caso il plugin abbia intenzioni poco "buone".

Per evitare queste spiacevoli situazioni è stato introdotto all'interno di PariPari un sistema di crediti, una sorta di "valuta" virtuale da spendere nel caso si voglia richiedere qualche servizio, oppure ottenibile fornendo un determinato servizio ad altri plugin.

Ogni plugin ha quindi un proprio "account", gestito ancora una volta dal *Core*, attraverso il "Credit System", che si occupa di depositare e prelevare i crediti da ogni account, lanciando `NotEnoughCreditsException` nel caso non vi siano nell'account i crediti necessari per pagare una certa operazione.

## 1.5 La rete

La rete si basa su un'architettura priva di server per varie ragioni. Con un'architettura distribuita non vi sono server per mantenere online la struttura, eliminando quindi la possibilità di un collasso totale della rete dovuto alla caduta di uno o più server. Secondo, ma non meno importante, è che questo tipo di architettura, a differenza dei server, non necessita di una manutenzione costante, portando ad un grande risparmio sia economico sia gestionale. Senza contare poi che la rete è meno vulnerabile ad attacchi di tipo *denial of service*, cosa non vera per una struttura basata su server.

# Capitolo 2

# Core

Il *Core* è il nocciolo centrale di ogni client di PariPari, si occupa infatti di far comunicare tra loro i plugin, provvedendo a caricare e spegnere i plugin stessi quando necessario e gestendo i crediti di ogni plugin. Oltre ai suoi compiti principali, il *Core* gestisce gli aggiornamenti, fornisce una GUI provvisoria e maneggia i thread presenti in PariPari.

## 2.1 Caricamento dei plugin

Una funzione importante del Core è proprio il caricamento dei plugin che si articola nei seguenti passaggi:

- Si crea una nuova istanza di `ClassLoader` con tutte le classi del plugin;
- Vengono creati tutti gli oggetti per comunicare con il Core e gli altri plugin;
- Gli oggetti appena creati vanno poi passati al costruttore del plugin;
- Viene avviato il thread principale del plugin.

Come specificato in precedenza, un plugin può dichiarare delle dipendenze verso altri plugin. Se, ad esempio, un plugin creato da terzi dovesse scrivere su disco i risultati delle sue elaborazioni dovrebbe delegare tale operazione al plugin `Local Store`, *dipendendo* di fatto da esso. Il Core carica le dipendenze in ordine, così non vi sono interruzioni durante il normale utilizzo di PariPari.

## 2.2 Comunicazione tra plugin

Sin dall'inizio il Core aveva il compito di gestire i messaggi tra plugin, i quali possono essere:

- Una richiesta singola ad un singolo plugin;
- Una richiesta multipla ad un singolo plugin;
- Una richiesta singola a più plugin;
- Una richiesta multipla a più plugin;

I passaggi effettuati da un messaggio sono riportati nella figura seguente, dalla quale prenderemo spunto per ripercorrere le varie fasi.
Nel nostro caso, basandoci dalla Figura 3, il plugin Alice vuole inviare un messaggio al plugin Bob. Come primo passo passa la richiesta al suo Mound Putter che successivamente viene inserita nel Mound principale, di proprietà del Core.
Il Mound è una coda di priorità i cui elementi vengono ordinati in base al tempo di attesa così da evitare problemi di mancata esecuzione nel caso una richiesta venga continuamente sorpassata da altre più prioritarie. Successivamente il Kernel si occupa di recuperare l'elemento più prioritario dal Mound per poi inserirlo nel Warehouse, una sorta di magazzino virtuale contenente tutte le richieste che devono essere soddisfatte.

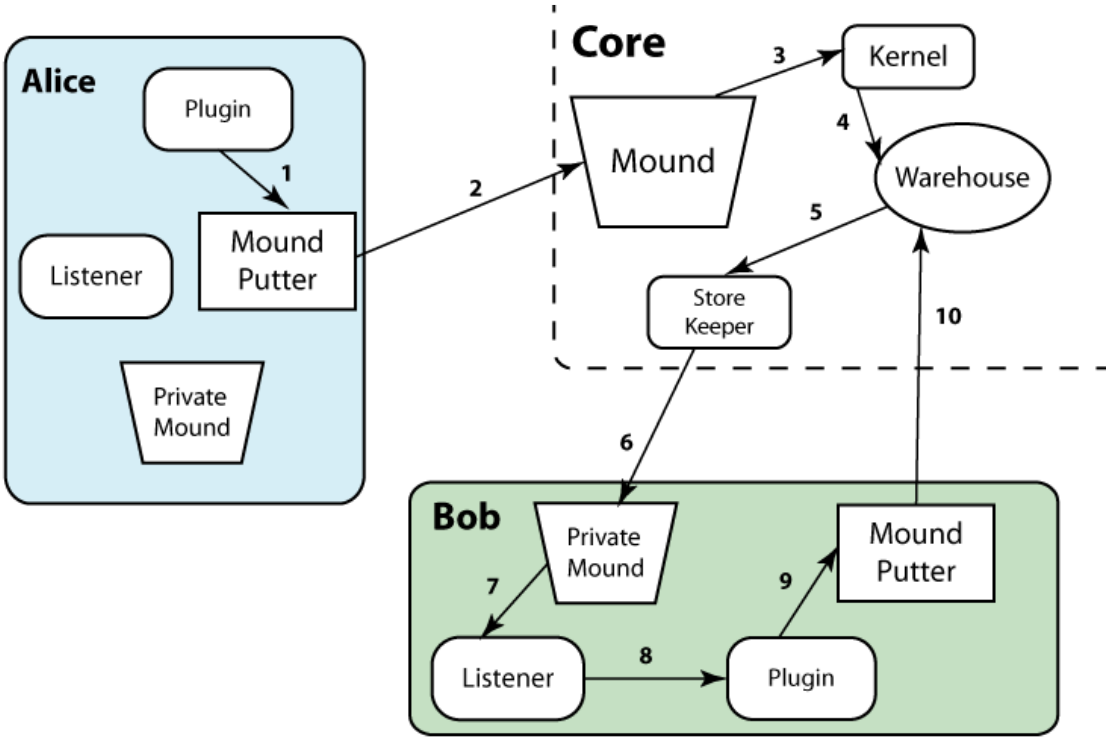


**Figura 3: Percorso di un messaggio tramite il Core**

Successivamente StoreKeeper ha il compito di prelevare gli ordini e smistarli verso i PrivateMound dei plugin destinatari, cercando di raggruppare messaggi con all'interno più richieste, anche da vari plugin, così da risparmiare tempo e memoria.

Sarà compito del Listener, uno per ogni plugin, di recuperare dal PrivateMound le richieste, rispondendo se necessario inviando un Reply al plugin richiedente, indicando se la richiesta è completa o meno ed eventualmente quante risorse sono state prodotte o recuperate.

Una volta completata la fornitura di risorse richieste, il plugin Alice verrà risvegliato controllando così la risposta inviatagli dal plugin Bob, comunicando al sistema di Crediti se il venditore, Bob nel nostro caso, ha fornito o meno le risorse richieste.

# Capitolo 3

# Il funzionamento del Timer

In questo capitolo tratteremo di come sono realizzati i Timer in Java e successivamente la loro implementazione in PariPari, analizzando il funzionamento e spiegando i motivi che hanno portato a questa soluzione.

## 3.1 I Timer in Java

La classe di riferimento è `java.util.Timer`[3]. Lo scopo di questa classe è quello di mettere a disposizione un modo semplice e affidabile per eseguire un insieme di task ad intervalli regolari a partire da un determinato istante, specificati durante la creazione.

Per fare ciò servirà un'istanza della classe `Timer` alla quale assegnare i task e i task stessi, che dovranno invece estendere la classe `java.util.TimerTask` e implementare nel metodo `run()` il codice da eseguire.

[3] http://download.oracle.com/javase/6/docs/api/index.html?java/util/Timer.html

Ad ogni istanza di `Timer` viene associato un thread di appoggio che ha il compito di eseguire i task sottomessi rispettando le tempistiche richieste ed eventualmente cercando di recuperare il tempo perso con rapide esecuzioni.
Un task deve essere completato velocemente, pena il rallentamento degli altri task presenti nella coda, con il conseguente "ammassamento" ed esecuzione in rapida successione per recuperare il tempo perduto.
Quando non vi è più alcuna reference al `Timer` e tutti i suoi task attivi hanno completato l'esecuzione, il thread associato al `Timer` termina silenziosamente, permettendo così all'utilizzatore di programmare l'esecuzione per poi dedicarsi ad altro.
Eventualmente il creatore del `Timer` può interrompere la sua esecuzione, nel caso non sia più richiesta o non più necessaria chiamando il metodo `cancel()`.
Da notare che nel caso il `Timer` venga cancellato, una successiva aggiunta di un task provocherà il lancio di `IllegalStateException`, non vi è quindi la possibilità di riutilizzare un `Timer` dopo che è stato cancellato, costringendo alla creazione di un nuovo Timer.

La classe `Timer` possiede quattro costruttori i cui argomenti sono (possono anche non essere specificati):

- `isDaemon` specifica che il thread associato al `Timer` deve essere di tipo *daemon*[4];
- `name` il nome da assegnare al thread.

Ogniqualvolta venga creato un nuovo `Timer` viene automaticamente creato il thread dedicato alla gestione della coda, rendendoli così indipendenti gli uni dagli altri.

Una volta creata la propria istanza di Timer sarò possibile schedulare un task attraverso i metodi `schedule()` o `scheduleAtFixedRate()` forniti dalla classe.

Vi è una sostanziale differenza in questi due metodi, infatti inserendo un task con il metodo `schedule()`, esso verrà eseguito dal momento richiesto rispettando l'intervallo di tempo tra le varie esecuzioni. Ma in caso di ritardo nell'esecuzione,

[4] Un thread di tipo daemon è un semplice thread di Java la cui esecuzione non impedisce alla JVM di terminare, ovvero la JVM termina nel momento in cui non ci sono thread non demoni attivi

dovuta a un fattore qualsiasi, le successive esecuzioni verranno a loro volta ritardate dello stesso tempo, così da ottenere un'esecuzione a *ritardo fisso* le une dalle altre.

Nel caso invece si voglia avere un'esecuzione con una *frequenza di esecuzione fissata* si può schedulare il task con il metodo `scheduleAtFixedRate()`. Il quale permette di "recuperare" il tempo perso, nel caso vi sia ritardo, eseguendo in rapida successione i task, così da riportarsi alla frequenza di esecuzione desiderata.

`Schedule` è disponibile con i seguenti parametri:

- `schedule(TimerTask task, Date firstTime)`: inserisce il task per essere eseguito una sola volta al tempo specificato;
- `schedule(TimerTask task, Date firstTime, long period)`: inserisce il task per un'esecuzione ripetuta nel tempo con un *ritardo fisso*, iniziando al tempo specificato;
- `schedule(TimerTask task, long delay)`: inserisce il task per una singola esecuzione dopo il ritardo specificato;
- `schedule(TimerTask task, long delay, long period)`: inserisce il task per un'esecuzione ripetuta nel tempo con un *ritardo fisso*, iniziando dopo il ritardo specificato;

Mentre `scheduleAtFixedRate` è presente solo con:

- `scheduleAtFixedRate(TimerTask task, Date firstTime, long period)`: inserisce il task un'esecuzione ripetuta nel tempo con un *frequenza di esecuzione fissata*, iniziando al momento specificato;
- `scheduleAtFixedRate (TimerTask task, long delay, long period)`: inserisce il task un'esecuzione ripetuta nel tempo con un *frequenza di esecuzione fissata* iniziando dopo il ritardo specificato;

### 3.1.1 TimerTask

I Timer in Java lavorano con oggetti di tipo `java.util.TimerTask`[5], classe astratta che implementa l'interfaccia `java.lang.Runnable`[6], il cui metodo `run()` dovrà essere implementato con il codice da eseguire.
La funzione più importante fornita dalla classe è la possibilità di cancellare l'esecuzione, attraverso il metodo `cancel()`.

## 3.2 I Timer in PariPari

La precedente implementazione di PariPariTimer riprendeva quanto visto in Java, modificando però alcune caratteristiche.
A differenza di Java infatti vi era una sola istanza di Timer alla quale tutti i plugin facevano affidamento.
Vi era inoltre una distinzione per i task periodici, essi infatti venivano gestiti separatamente, lanciando un thread per ogni task periodico incaricato all'esecuzione programmata.

La nuova implementazione di PariPariTimer differisce in diversi punti rispetto a quella di Java e a quella precedente: essa riprende il concetto utilizzato in Java, in cui molti Timer risultano attivi e programmano l'esecuzione dei loro task, ma allo stesso tempo segue la via dettata dalla precedente versione del Timer di PariPari, l'idea di una gestione centralizzata dei task, realizzata dalla classe `PariPariCoreTimer`.
La classe `paripari.core.PariPariCoreTimer` di fatto diventa il nuovo Timer, disponendo della coda di esecuzione dei task e del thread dedicato alla gestione ed esecuzione dei task, rimanendo però dietro le quinte e raggiungibile direttamente solo dalle classi appartenenti al package del Core.
La classe `paripari.core.PariPariTimer` è ancora presente, ma il suo funzionamento è stato modificato rispetto alla versione precedente, perdendo di fatto la

[5] http://download.oracle.com/javase/6/docs/api/java/util/TimerTask.html
[6] http://download.oracle.com/javase/6/docs/api/java/lang/Runnable.html

gestione dei task e limitandosi a ruolo di intermediario con la nuova classe addetta a ruolo di gestore.

La classe `TimerTask` di Java è stata sostituita dall'interfaccia `paripari.core.interfaces.PariPariTimerTask` che semplicemente estende l'interfaccia `paripari.core.interfaces.PariPariRunnable`, perdendo di fatto i metodi `cancel()` e `scheduledExecutionTime()`.

Cambiano anche i task gestiti dal Timer, non sono più estensioni della classe `java.util.TimerTask`, ma bensì degli oggetti proxy realizzati dalla classe `paripari.core.PluginTask`.

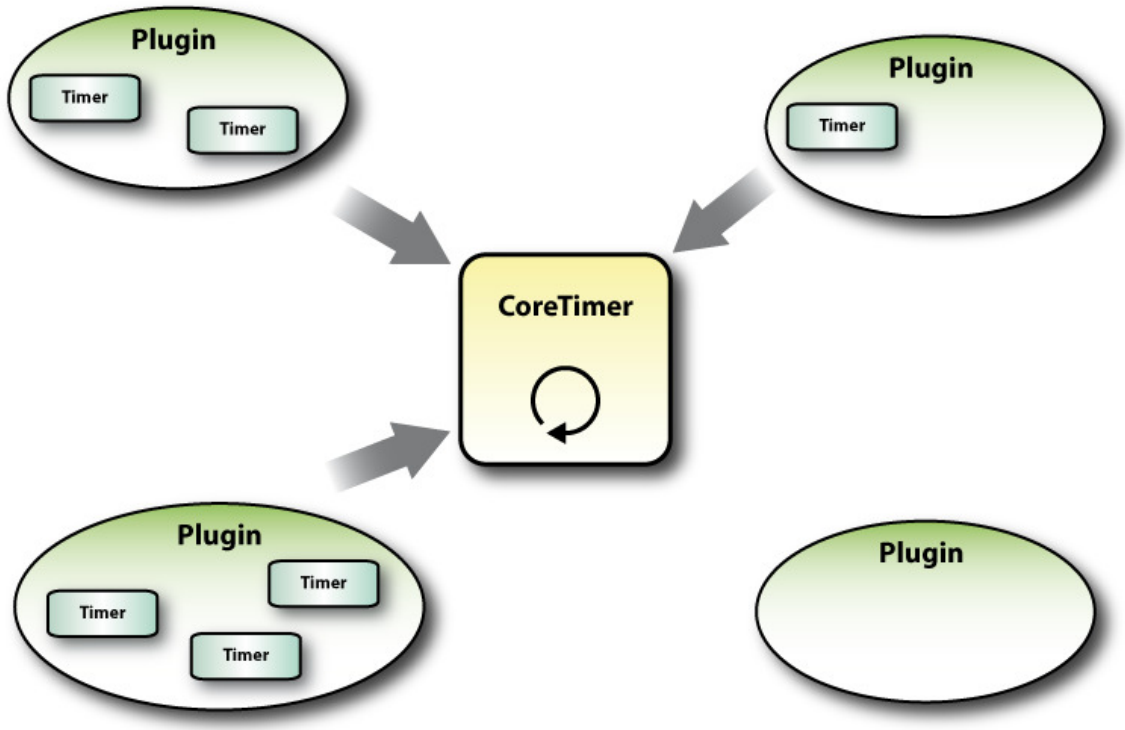


**Figura 4: Collegamento tra CoreTimer e Timer appartenenti ai Plugin**

Tutti i Timer sono registrati presso il CoreTimer al momento della loro creazione, inseriti in una mappa che collega Plugin ai rispettivi Timer, permettendo così di rintracciare semplicemente il proprietario.

### 3.2.1 PariPariCoreTimer, il nuovo motore

Come accennato poco prima, la gestione dei task di PariPari è effettuata dalla classe `paripari.core.PariPariCoreTimer`.

Non è però possibile istanziare multiple istanze di `CoreTimer` (il suo costruttore è `private`), si può solo ottenere l'accesso all'unica istanza disponibile. Il `CoreTimer` infatti sfrutta il *Design Pattern Singleton*[7] per avere una sola copia attiva di se stesso, evitando al contempo che altri possano creare altre istanze.

Per fare ciò si impone che i costruttori della classe siano di tipo `private`, quindi inaccessibili dall'esterno, si fa in modo che venga creata una sola istanza di esso durante il class loading situata in una variabile privata di tipo `final`, ovvero non modificabile:

```
private static final PariPariCoreTimer TIMER = new
PariPariCoreTimer();

protected static PariPariCoreTimer getPariPariCoreTimer() {
            return TIMER;
    }
```

E' inoltre possibile fermare l'esecuzione del `CoreTimer`, cancellando l'esecuzione di tutti i task inseriti ed eliminando i collegamenti con i plugin attraverso i metodi `cancel()` e `cancel(long remainingTime)`.

Entrambi i metodi fanno affidamento ad un task vuoto (senza istruzioni da eseguire) impostato come task di terminazione tramite la chiamata al metodo `stoppingTask()` presente nella classe `PariPariPluginTask`.

La chiamata a questo metodo fa sì che alla successiva chiamata di `isStopRequest()` effettuata sul task, essa restituisca il valore `true`, terminando così l'esecuzione.

Eventualmente è possibile richiedere che la terminazione non sia immediata specificando quanto tempo (`long remainingTime` , in millisecondi) lasciare di esecuzione prima di interrompere il ciclo di esecuzione dei task.

[7] http://en.wikipedia.org/wiki/Singleton_pattern

### 3.2.1.a Programmare l'esecuzione di un task

Quando un generico `Timer` richiede l'esecuzione di un task (una sua implementazione dell'interfaccia `TimerTask`) attraverso uno dei suoi metodi `schedule()` o `scheduleAtFixedRate()`, il `CoreTimer` prende in carico il task (dopo essere stato incapsulato in un oggetto `PluginTask`, il quale verrà discusso in seguito).

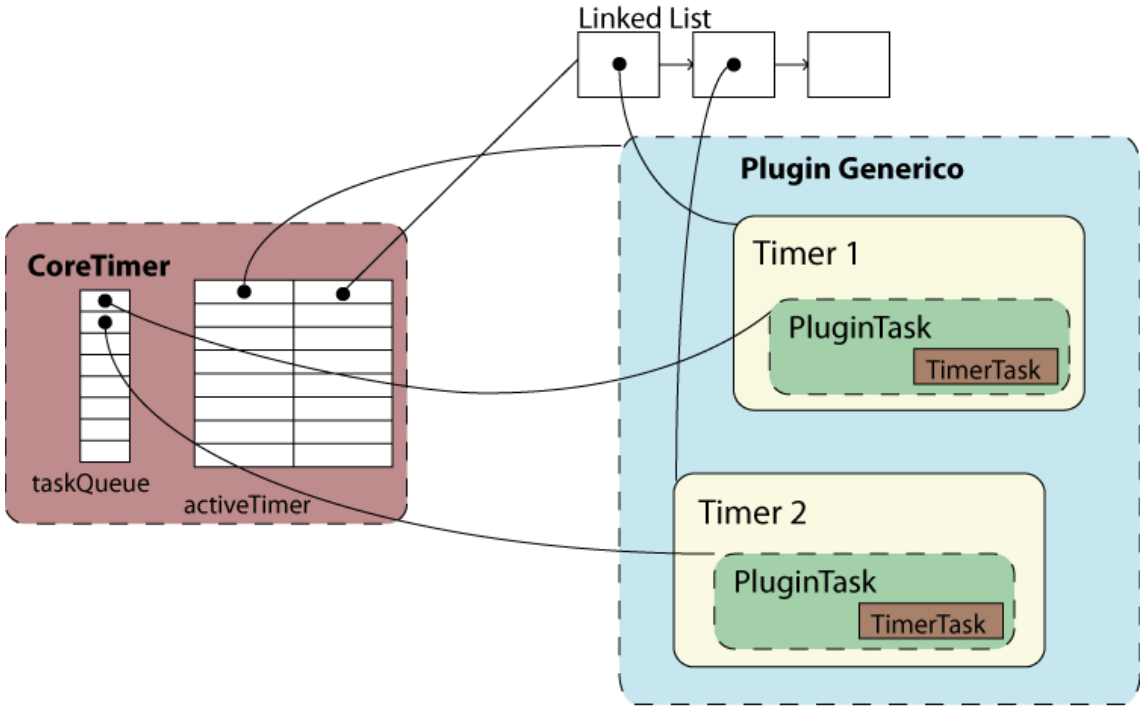


**Figura 5: Attivazione di un task**

Viene verificato che il task non sia già stato schedulato mentre i suoi parametri vengono controllati per evitare il lancio di eccezioni o comportamenti anomali. Una volta controllati vengono infine inseriti negli appositi campi di un oggetto `PariPariPluginTask` assieme all'implementazione di `PariPariTimerTask` fornita dal Timer. Come ultimo passaggio l'oggetto `PariPariPluginTask` viene inserito nella coda di priorità presente nel `CoreTimer` in attesa di essere eseguito.

### 3.2.1.b PariPariTaskQueue e DelayQueue, la coda di esecuzione

La classe `PariPariTaskQueue`, presente anche nella precedente versione di Paripari, è incaricata di gestire la coda di task.

Nella precedente versione era però realizzata con un array di oggetti `PariPariTimerTask` e andava gestita manualmente per mantenere il corretto ordinamento. Ora invece la classe si limita a interagire con la coda `taskQueue` presente al suo interno, inserendo e rimuovendo da essa gli elementi.

Sono state effettuate queste modifiche per sfruttare la classe `java.util.concurrent.DelayQueue<E>` presente dalla versione 1.5 di Java. Questa classe infatti semplifica enormemente la gestione della coda di priorità, lasciando a noi solo il compito di richiedere oppure inserire i task quando necessario.

La coda si basa su una struttura ad heap ed è realizzata da un'implementazione dell'interfaccia `java.util.concurrent.BlockingQueue` [8] di dimensioni illimitate. Essa si basa sulla struttura di una normale `Queue`, aggiungendo metodi che attendono finché la coda diventa non vuota quando si cerca di recuperare un elemento e attendono fino a quando si libera dello spazio disponibile, nel caso la coda abbia una dimensione limitata.

Non avendo limiti di dimensione (se non quelli della dimensione dell'heap della JVM), i metodi di inserimento quali `add(E e)`, `offer(E e)` e `put(E e)` inseriscono immediatamente l'oggetto in coda, senza bloccarsi a differenza delle `BlockingQueue`, ma lanciando `NullPointerException` se viene passato `null` come parametro, `DelayQueue` infatti non accetta valori `null`. Sempre la coda si occupa poi di riordinare gli elementi al suo interno confrontandoli tra loro con il metodo `compareTo()`.

[8] http://download.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html

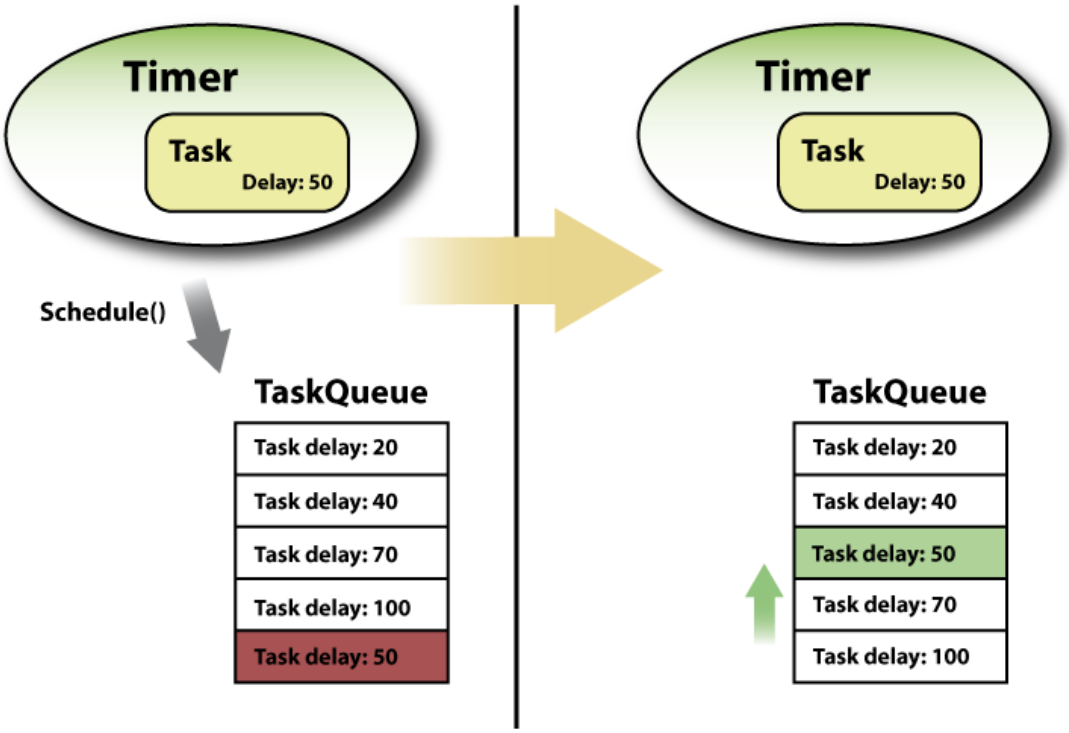


**Figura 6: Funzionamento interno della coda DelayQueue**

Va specificato che gli oggetti presenti nella coda, oggetti di tipo `paripari.core.PariPariPluginTask`, dei quali parleremo più avanti, sono oggetti che implementano l'interfaccia `Delayed` del pacchetto `concurrent` di Java. L'implementazione dell'interfaccia porta con se il compito di implementare due metodi, `compareTo(Delayed others)` e il metodo `getDelay(TimeUnit unit)` utilizzati per confrontare gli oggetti tra loro. Proprio il metodo `compareTo(Delayed others)` è quello utilizzato per confrontare gli oggetti tra loro all'interno della coda.
I metodi di recupero degli elementi quali `peek()`, `poll()` e `take()` agiscono diversamente. Se i primi due ritornano un valore `null` quando la coda è vuota, nel primo caso, o quando non vi sono elementi con ritardo scaduto, nel secondo caso, il metodo `take()` fa attendere finché non vi è un elemento il cui ritardo è scaduto.

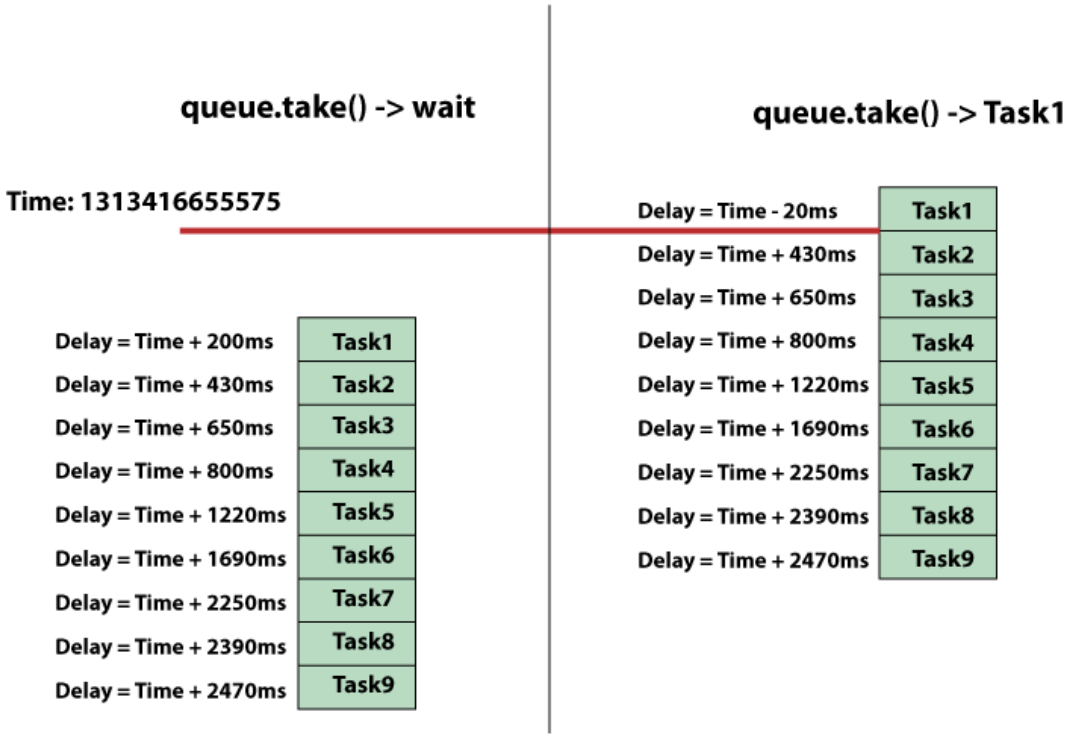


**Figura 7: Funzionamento della coda DelayQueue**

Proprio il metodo `take()` è stato sfruttato all'interno del `mainLoop()` del thread gestore della coda, evitando così di inserire un ciclo `while(true)` presente nella precedente versione.

### 3.2.1.c ThreadQueueHandlerRunnable, il gestore della coda

Mentre la coda si preoccupa di fornire al momento opportuno i task, la loro esecuzione e gestione è nelle mani di un'altro componente di `CoreTimer`, il thread `ThreadQueueHandlerRunnable`.

Il suo compito è quello di recuperare i task pronti per l'esecuzione dalla coda `taskQueue`, programmare, se necessario, la loro successiva esecuzione ed eseguire effettivamente il codice presente nel task.

```
private void mainLoop() {
try {
PariPariPluginTask task = queue.getRemoveMin();
while (!(task.isStopRequest())){
synchronized (task.getLock()) {
    // don't execute task already executed or canceled
    if (task.getState() != PariPariPluginTask.TaskState.CANCELED.getState()
            && task.getState() != PariPariPluginTask.TaskState.EXECUTED.getState()) {
        if (task.getPeriod() == 0) {  // non-periodic task
            task.setState(PariPariPluginTask.TaskState.EXECUTED);
            // execute single task
            executeTheTask(task);
        } else {
            // periodic task
            if (task.getPeriod() < 0) { // SCHEDULE
                task.setNextExecutionTime(System.currentTimeMillis() - task.getPeriod());
            }
            else { // SCHEDULE AT FIXED RATE
                task.setNextExecutionTime(task.getNextExecutionTime() + task.getPeriod());
            }
            // check if is necessary to re-schedule the task
            if (task.getEndTime() > task.getNextExecutionTime() || task.getEndTime() == -1){
                queue.add(task);
            }
            else{
                task.setState(PariPariPluginTask.TaskState.EXECUTED);
            }
            // check completed, execute the task
            executeTheTask(task);
        }
    }
}
task = queue.getRemoveMin();
} // while end
}
catch (InterruptedException e) {
    // do nothing
}
}
```

I passaggi in dettaglio sono:

1. Richiede alla coda se vi sono task da eseguire tramite il metodo `queue.getRemoveMin()`, rimanendo in attesa fino a quando la coda non restituisce un task;
2. Verifica che il task ricevuto dalla coda non sia una richiesta di interruzione ,tramite la chiamata al metodo `isStopRequest()`, altrimenti termina la sua esecuzione senza lanciare eccezioni (questa parte verrà descritta con più attenzione nella sezione successiva);
3. Ottiene il lock sul task in modo da evitare modifiche concorrenti allo stesso task;

4. Verifica che il task non sia stato precedentemente cancellato od eseguito, onde evitare di eseguire codice che non era in programma;
5. Controlla se il task era programmato per la singola esecuzione o quella periodica, differenziando i due casi:
   5.1. *Singola esecuzione*: imposta lo stato del task come EXECUTED (gli stati sono definiti dall'Enum `TaskState` presente in `PariPariPluginTask`, maggiori approfondimenti in seguito);
   5.2. *Esecuzione periodica*: come prima cosa calcola il tempo della prossima esecuzione programmata, sommando il tempo corrente (in millisecondi) al periodo del task nel caso dell'esecuzione programmata con *ritardo fisso*. Nel caso della *frequenza di esecuzione fissata* invece, il tempo della prossima esecuzione verrà calcolato basandosi sul tempo dell'ultima esecuzione;
6. Successivamente verifica se il task, solo nel caso periodico, non ha ancora terminato la sua esecuzione, controllando se il tempo della prossima esecuzione programmata è antecedente quello del tempo di fine esecuzione. Se la condizione è verificata re-inserisce il task nella coda di esecuzione. Se la condizione non viene soddisfatta imposta lo stato del task come EXECUTED;
7. Come ultimo passaggio, solo nel caso periodico, lancia il codice eseguibile contenuto nel task e si prepara a eseguire nuovamente il ciclo.

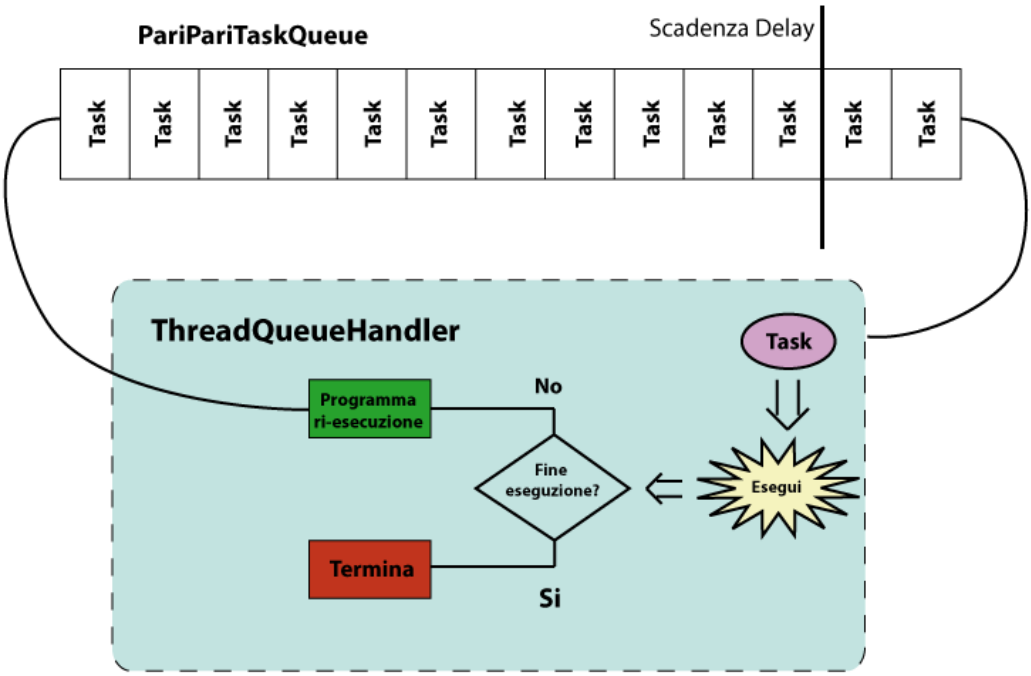


**Figura 8: Esecuzione ThreadQueueHandlerRunnable**

### 3.2.2 PariPariTimer, la nuova implementazione

Come spiegato in precedenza la classe `Timer` di Java è stata sostituita dalla classe `PariPariTimer` che presenta sostanziali differenze rispetto alla realizzazione in Java.

Prima di tutto le istanze di Timer non sono a sé stanti, ma dipendono dal `CoreTimer`, quindi i costruttori perdono la possibilità di specificare che il thread dedicato alla gestione del Timer sia di tipo *daemon*. Questo è dovuto al fatto che le istanze di Timer non hanno un loro thread associato, dato che viene tutto gestito dal `CoreTimer`.

E' ancora possibile specificare il nome del Timer, ma nel caso non venga specificato verrà assegnato un nome riprendendo il nome del Plugin richiedente.

I costruttori quindi si riducono da quattro a due e sono i seguenti:

- `PariPariTimer()`: avvia un nuovo Timer, registrandolo al `CoreTimer`, pronto per l'esecuzione di codice;

- `PariPariTimer(String name)`: avvia un nuovo Timer con il nome specificato come parametro, registrandolo al `CoreTimer` come nel caso precedente.

I metodi di inserimento di task invece rimangono sostanzialmente invariati rispetto alla precedente versione, salvo differire in un paio di occasioni rispetto alla versione in Java. Come Java permettono di programmare l'esecuzione per un *ritardo-fisso* oppure per una *frequenza di esecuzione fissata*, a seconda delle necessità, introducendo al contempo la possibilità di specificare un tempo di fine esecuzione.

Nello specifico, per quanto riguarda la programmazione a *ritardo-fisso*:

- `schedule(PariPariTimerTask task, Date firstTime)`
- `schedule(PariPariTimerTask task, Date firstTime, long period)`
- `schedule(PariPariTimerTask task, Date firstTime, long period, Date endTime)`
- `schedule(PariPariTimerTask task, Date firstTime, long period, long endTime)`
- `schedule(PariPariTimerTask task, long delay)`
- `schedule(PariPariTimerTask task, long delay, long period)`
- `schedule(PariPariTimerTask task, long delay, long period, long endTime)`

Mentre per la controparte a *frequenza di esecuzione fissata*, abbiamo:

- `scheduleAtFixedRate(PariPariTimerTask task, Date firstTime, long period)`
- `scheduleAtFixedRate(PariPariTimerTask task, Date firstTime, long period, long endTime)`
- `scheduleAtFixedRate(PariPariTimerTask task, long delay,long period, Date endTime)`

- `scheduleAtFixedRate(PariPariTimerTask task, long delay, long period)`
- `scheduleAtFixedRate(PariPariTimerTask task, long delay, long period, long endTime)`

La versione attuale del Timer, a differenza di Java e della precedente versione di PariPari, deve introdurre un paio di metodi per cancellare l'esecuzione di un determinato task (dovuto al fatto che ora l'interfaccia `TimerTask` non mette a disposizione un metodo per cancellare l'esecuzione) e introduce la possibilità di cancellare un determinato Timer, quindi interrompere l'esecuzione di tutti i suoi task.
Per fare ciò sfrutta una mappa che associa implementazioni di interfaccia `TimerTask` e la loro controparte `PluginTask`. La mappa è di tipo `WeakHashMap`[9] , ovvero una `HashMap` le cui chiavi sono di tipo `Weak`, deboli. Questo sta ad indicare che le chiavi sono in realtà degli oggetti di tipo `WeakReference` che puntano agli oggetti `TimerTask` passati alla mappa.
Un oggetto di tipo `WeakReference`[10] è un oggetto che non previene gli oggetti riferiti dall'essere finalizzati, e quindi reclamati dal *Garbage Collector*. Si ha che, quando il *Garbage Collector* determina che un oggetto diventa *debolmente raggiungibile* rimuoverà atomicamente tutte le reference deboli all'oggetto. Allo stesso tempo dichiarerà che tutti gli oggetti precedentemente *debolmente raggiungibili* vengano finalizzati.
Si è scelto di usare questo tipo di HashMap per limitare il numero di oggetti e quindi lo spreco di memoria. Una voce della mappa viene rimossa appena l'oggetto riferito non esiste più o sta per essere eliminato. Nel nostro caso quindi le voci vengono eliminate appena un plugin termina oppure elimina i riferimenti alle istanze `TimerTask` che ha schedulato al proprio `Timer`.

La mappa serve principalmente al `Timer` per tenere traccia di quali e quanti `TimerTask` sono stati attivati permettendo poi di accedere agli stessi nel caso vi sia la necessità, come nel caso si voglia cancellare l'esecuzione di un particolare task.

[9] http://download.oracle.com/javase/6/docs/api/java/util/WeakHashMap.html
[10] http://download.oracle.com/javase/6/docs/api/java/lang/ref/WeakReference.html

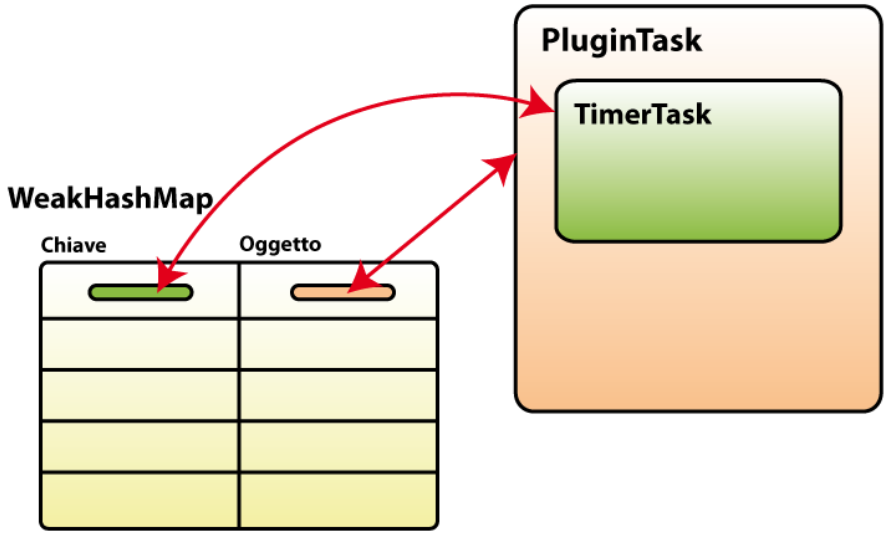


**Figura 9: Utilizzo WeakHashMap presente in ogni Timer**

Infatti, come specificato precedentemente, non è più possibile cancellare un task chiamando il suo metodo `cancel()` dato che esso non è più presente. Tale metodo però è stato implementato nel Timer che quindi offre due tipi di metodi `cancel()`:

- `cancel()`, cancella completamente il Timer, cancellando tutti i suoi task, se presenti, ed eliminando il Timer dalla lista di timer attivi presente nel `CoreTimer` bloccando al contempo la possibilità di schedulare altri task su tale Timer, di fatto invalidando definitivamente il Timer. L'operazione non è reversibile;
- `cancelTask(PariPariTimerTask task)`, cancella l'esecuzione del task passato come parametro, se presente nella coda, fallendo silenziosamente in caso contrario.

Vi è stato aggiunto un ulteriore metodo, meno drastico del `cancel()` ma con una funzione simile. Il metodo `dropScheduledTasks()` cancella l'esecuzione di tutti i task presenti nel Timer, ma non elimina il Timer stesso dalla lista di timer attivi

presente nel `CoreTimer`, lasciando così la possibilità di assegnare altri task senza dover creare un nuovo Timer.
A completare la dotazione di metodi è stato aggiunto il metodo `allDone()` che ritorna il valore booleano `true` se tutti i task inseriti su questo timer hanno completato la loro esecuzione.
Come il Timer di Java, quando non vi sono più reference al Timer e i task attivi sono stati completati, il Timer silenziosamente.

### 3.2.3 PariPariPluginTask, il sostituto di TimerTask

Come specificato in precedenza, il Timer non lavora più con elementi che estendono la classe `TimerTask` di Java, ma non può lavorare solo con elementi che realizzino l'interfaccia `TimerTask` di PariPari dato che essa non mette a disposizione tutti i metodi necessari per gestire correttamente l'esecuzione dei task.
E' stato quindi necessario introdurre una classe che inglobasse l'implementazione dell'interfaccia fornita dai plugin al suo interno, fornendo al contempo i metodi necessari alla gestione dei task all'interno del `CoreTimer`.
A questo scopo è nata la classe `paripari.core.PariPariPluginTask` che implementa i metodi `cancel()` e `scheduledExecutionTime()` della classe `TimerTask` di Java e allo stesso tempo implementa i metodi necessari per l'utilizzo di questa classe all'interno della `DelayedQueue` del `CoreTimer`, quali `compareTo()` e `getDelay()`.
La classe infatti implementa le interfacce `java.util.concurrent.Delayed` e `java.land.Runnable`, la prima perché, come introdotto precedentemente, la coda `DelayedQueue` richiede oggetti che implementino l'interfaccia `Delayed`, necessario per confrontare gli oggetti tra di loro così da poter stabilire le tempistiche di esecuzione, la seconda per eseguire il codice all'interno del metodo `go()` implementato nell'interfaccia.

I metodi richiesti dall'interfaccia, e la loro implementazione, `Delayed` sono i seguenti:

```
public long getDelay(TimeUnit unit){
   return unit.convert(nextExecutionTime - System.currentTimeMillis() -
c,unit);
```

```
}

public int compareTo(Delayed other) {
if (getDelay(TimeUnit.MILLISECONDS)==other.getDelay(TimeUnit.MILLISECONDS))
   return 0;
else
   return getDelay(TimeUnit.MILLISECONDS) > other.getDelay(TimeUnit.MILLISECONDS) ? 1 : -1;
}
```

Il primo metodo `getDelay()`, come specificato dal nome, restituisce il ritardo rimanente prima della prossima esecuzione programmata del task chiamante sottraendo il tempo programmato per la prossima esecuzione dal tempo attuale.

Il metodo `compareTo()` invece compara due istanze di oggetti `Delayed`, stabilendo quale dei due viene prima o eventualmente se hanno egual ritardo, restituendo 0 se le due istanze sono programmate per essere lo stesso istante, 1 se il task chiamante ha ritardo maggiore del task passato come parametro, -1 viceversa.

Per quanto riguarda l'interfaccia `Runnable` di Java vi è solo il metodo `run()` da implementare. L'implementazione attuale è:

```
public void run() {
      try {
            myTask.go();
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
```

La classe però non può essere istanziata direttamente dai plugin, essendo di tipo *package-protected*, viene infatti istanziata dal Timer del plugin nel momento in cui esso schedula un task per l'esecuzione, aggiungendo ad esso l'implementazione di `TimerTask` fornita dal plugin stesso.

Nel paragrafo 3.2.1.c si era parlato del metodo `mailLoop()` appartenente al thread dedicato alla gestione della coda e di come la sua esecuzione si fermasse una volta ricevuto un task il cui metodo `isStopRequest()` restituisca il valore `true`.

Ogni `PluginTask` infatti dispone di due metodi, `stoppingTask()` e `isStopRequest()` volti a trasformare un normale `PluginTask` in uno `StoppingTask`.

I metodi sono però entrambi protetti e mentre il secondo viene chiamato all'interno del `mainLoop()`, il secondo viene chiamato all'interno dei metodi `cancel()` o `cancel(long remainingTime)` del `CoreTimer`.
Il metodo crea un task il cui metodo *go* non ha codice (dato che comunque non verrebbe eseguito), ma viene impostato come *stoppingTask* chiamando l'omonimo metodo prima di inserire il task nella coda di esecuzione.
E' inoltre possibile specificare eventualmente un tempo di terminazione, così da lasciare un certo tempo per eseguire gli ultimi task rimanenti prima di terminare il processo di esecuzione.

# Conclusioni

L'utilizzo di classi fornite dalle librerie standard Java ha permesso di semplificare la gestione dei Timer. La sola classe `DelayQueue` di Java si occupa di fornire in uscita al momento opportuno i task che devono essere eseguiti, sollevandoci dall'incarico di controllare istante per istante quali task dovevano essere eseguiti. La mappa `WeakHashMap` dal canto suo evita lo spreco di memoria senza che il programmatore debba preoccuparsi di pulire la mappa di tanto in tanto.
Il refactoring ha permesso poi una gestione più semplice anche da parte dei plugin. Ora possono avere più istanze di Timer, ognuna con i suoi `TimerTask` in esecuzione permettendo come in Java di cancellare un singolo Timer, oppure cancellare solo i task schedulati, permettendo di riutilizzare il Timer a differenza di Java.